Riassunto modulo 1 (A.A 2019-2020)

Fasi del processo di sviluppo del software

Requisito: descrizione di

- un comportamento atteso dal sistema (requisito funzionale)
- un vincolo sul comportamento del sistema o sullo sviluppo del sistema (requisito non funzionale)

Lo sviluppo del software si divide in più fasi:

1. **Studio di fattibilità**: consiste in una valutazione preliminare dei costi e dei benefici, che porta ad una definizione preliminare del problema e ad una descrizione di costi e tempi necessari
2. **Analisi dei requisiti**: prevede una formalizzazione dei requisiti e una successiva analisi che porta alla redazione di un documento di specifica (da validare da parte del committente). Il documento deve specificare quali siano le funzionalità senza dire come verranno realizzate e rappresenta un contratto tra sviluppatori e committente
3. **Analisi del problema**: dalla lettura del documento dei requisiti si descirve l'architettura logica del sistema basata sul modello del dominio. L'architettura logica del sistema deve esprimere fatti il più possibile oggettivi sul problema (non sulla sua soluzione) focalizzando l'attenzione sui sottosistemi, sui ruoli e sulle responsabilità. In sostanza è un modello che descrive la struttura del sistema, il comportamento atteso e le interazione tra le varie parti
4. **Progettazione**: l'architettura del sistema è scomposta in più programmi eseguibili (ognuno descritto dalle sue funzionalità e dall'interazione che ha con gli altri programmi), a loro volta scomposti in moduli (ognuno caratterizzato dalle proprie funzioni e relazioni con gli altri moduli)
5. **Realizzazione e collaudo dei moduli**
6. **Integrazione e collaudo del sistema**
7. **Installazione e training**
8. **Utilizzo e manutenzione**: la manutenzione può puntare a correggere degli errori non rilevati in fase di collaudo, può mirare ad aggiungere nuovi servizi a quelli già esistenti, oppure a migliorare le caratteristiche già esistenti

Fattori di qualità del software

- **Correttezza**: data una definizione dei requisiti che il software deve soddisfare, il software si dice corretto se rispetta tali requisiti
- **Robustezza**: il software si dice robusto se si comporta correttamente (o in maniera accettabile) anche in presenza di anomalie od errori. In questi casi si può optare per una chiusura controllata del software senza perdite di dati
- **Affidabilità**: un software si dice affidabile se e solo se è sia corretto che robusto
- **Facilità d'uso**
- **Efficienza**
- **Estensibilità**: facilità con cui il software può essere modificato. Per migliorarla è meglio sfruttare un'architettura semplice ed una divisione in moduli autonomi
- **Riusabilità**
- **Verificabilità**: ovvero la facilità con cui il software può essere sottoposto ai test
- **Portabilità**

Principi OO

Programmazione basata sugli oggetti

Si introducono gli **ADT** (dati + codice). Ogni ADT ha un'interfaccia visibile all'esterno tramite cui è possibile accedere ai dati (o ad eventuali funzionalità) che invece sono nascosti. A livello teorico si parla di **information hiding** che viene implementato tramite l'**incapsulamento** ossia la separazione della parte pubblica e della parte relativa all'implementazione nascondendo le scelte progettuali (spesso mutevoli), minimizzando le modifiche necessarie in fase di manutenzione e migliorando la riusabilità del codice.

Oggetti e classi

Un **oggetto** è un'entità del programma che possiede:

1. uno stato (inteso come insieme di valori associati ai suoi attributi)
2. un insieme di operazioni che operano sullo stato e forniscono servizi ad altri oggetti
3. un comportamento
4. una identità univoca

Gli oggetti sono raggruppabili in **classi**, ossia entità che descrivono oggetti con caratteristiche (attributi e operazioni) comuni. A *compile time* una classe definisce l'implementazione di un ADT, mentre a *runtime* ogni oggetto è istanza di una classe.

Programmazione orientata agli oggetti

Polimorfismo

Può essere definito come la capacità della stessa cosa di apparire in forme differenti o di cose differenti di apparire nella medesima forma. Secondo la classificazione Cardelli-Wagner ci sono quattro tipi di polimorfismo:

- Universale
    - per inclusione: (resa possibile dall'ereditarietà di interfaccia e di estensione) porta alla realizzazione dell'overriding dei metodi (astratti e concreti) e del binding dinamico mediante la Virtual Method Table (programmazione OO)
    - parametrico: porta alla programmazione generica rispetto ai tipi
- Ad hoc
    - oveloading: stesso nome ma diversa implementazione a seconda del tipo di parametri: es, 23+4 (int, int): int 12.34 + 1.0 (float, float) : float
    - coercion: promozione automatica di tipi: es. 12.34 + 1 (1 viene convertito in float)

Ereditarietà

Le classi possono essere organizzate in una gerarchia di **ereditarietà**. È importante che gli oggetti della sottoclasse esibiscano tutti i comportamenti e le proprietà della superclasse (**Principio di sostituibilità di Liskov**), ma una sottoclasse può anche esibire comportamenti aggiuntivi o eseguire in maniera differente alcune funzionalità della superclasse.
L'ereditarietà migliora l'estendibilità del software in quanto evita di dover replicare codice e permette di ridefinire solamente le caratteristiche specifiche all'interno di una gerarchia.

Esistono più tipi di ereditarietà:

- ereditarietà di modello: riflette una relazione "Is a". Rappresenta un meccanismo di compatibilità tra tipi lungo una gerarchia
    - **ereditarietà di interfaccia**
    - **ereditarietà di estensione**
- ereditarietà software: rappresenta relazioni all'interno del software stesso anziché all'interno del modello
    - **ereditarietà di realizzazione**: rappresenta un meccanismo di riuso all'interno del software. Usata spesso quando una classe ha bisogno di utilizzare i servizi forniti da un'altra classe. La superclasse definisce parte dell'implementazione della sottoclasse e quindi si rompe l'incapsulamento rendendo più difficile il riuso della sottoclasse.
    - **composizione e delega**: anziché far derivare la classe B dalla classe A di cui sono necessari i servizi si crea un attributo privato all'interno di B avente tipo A e si delega a quell'oggetto lo svolgimento del compito. Con questa tecnica inoltre il legame avviene a runtime garantendo maggiore flessibilità.

Inoltre si hanno:

- **ereditarietà semplice**: ogni classe deriva da una e sola superclasse. Si ottiene così una struttura ad albero. Questa tecnica è usata da Java e .NET ad esempio

- **ereditarietà multipla**: almeno una classe deriva da più superclassi (esempio C++). La struttura che si ottiene in generale è un reticolo. Questo tipo di ereditarietà porta con sé possibili conflitti di nome tra attributi e metodi ereditati dalle varie classi. Tra le classi di questo tipo di gerarchia possono esistere i vincoli *overlapping* e *disjoint*. È sempre possibile riportarsi ad un caso di ereditarietà semplice mediante l'utilizzo di "composizione e delega" o tramite l'uso di interfacce.

Ereditarietà dinamica

L'ereditarietà dinamica ha senso quando abbiamo necessità di rappresentare un oggetto di cui son presenti più tipologie differenti che hanno caratteristiche particolari rispetto alla classe generale. Due metodi per simulare l'ereditarietà dinamica sono:

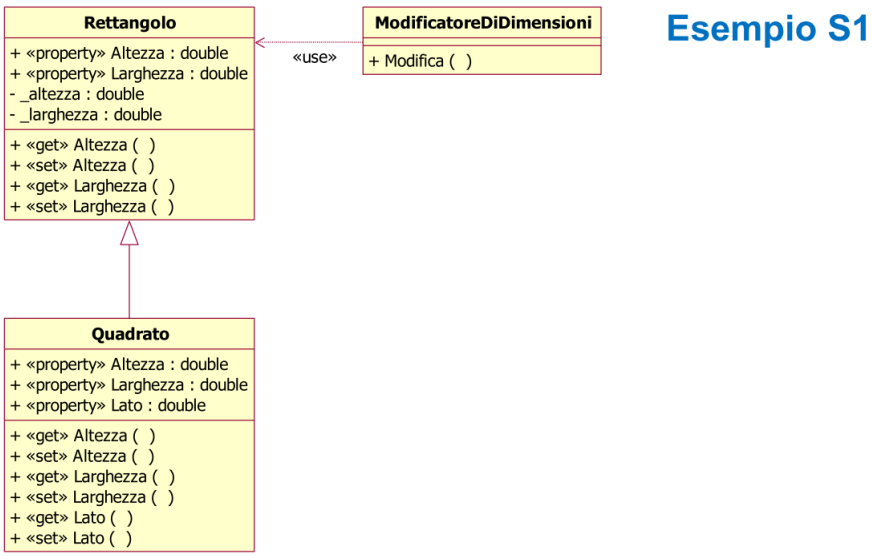
- Usare una factory che, a seconda dei parametri di ingresso forniti, costruisca l'istanza della classe corretta. Questo metodo pone un problema nel caso in cui i parametri della classe sono modificabili e dunque una modifica degli stessi potrebbe causare un cambiamento della classe.
- Usare il pattern State per risolvere il problema descritto sopra. Tale pattern, attraverso il meccanismo di composizione e delega, permette di incorporare l'istanza del tipo all'interno della classe generale; nel caso in cui avvenga una modifica ai parametri si controlla se occorre cambiare il tipo della classe e a seconda del tipo saranno accessibili una serie di parametri e metodi propri di quel caso particolare

Principio di sostituibilità di Liskov

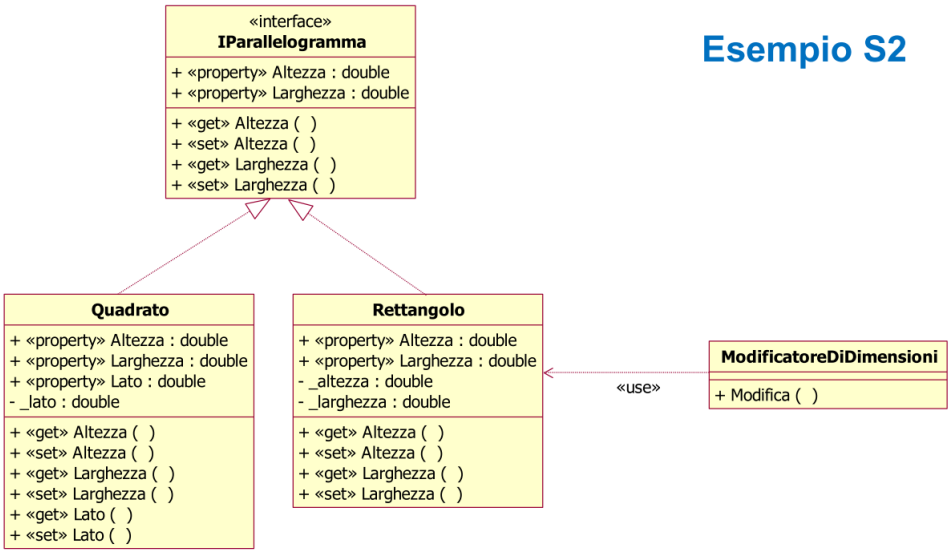
Una sotto-classe deve sempre essere una versione più specializzata della sua super-classe. Un buon test sul corretto utilizzo dell'ereditarietà è che sia valido il principio di sostituibilità di Liskov: "B è una sotto-classe di A se e solo se ogni programma che utilizzi oggetti di classe A può utilizzare oggetti di classe B senza che il comportamento logico del programma cambi".

Perché ciò sia valido, è necessario che:

- le pre-condizioni di tutti i metodi della sotto-classe siano uguali o più deboli
- le post-condizioni di tutti i metodi della sotto-classe siano uguali o più forti
- ogni metodo ridefinito nella sotto-classe deve mantenere la semantica del metodo originale



Nell'esempio S1 notiamo che il quadrato ha un vincolo in più rispetto al rettangolo: altezza e larghezza devono essere uguali. Il metodo Modifica della classe ModificatoreDiDimensioni funziona correttamente su un Rettangolo ma non funziona correttamente su un Quadrato. Quindi non è possibile passare un'istanza di Quadrato dove è prevista un'istanza di Rettangolo (il principio di sostituibilità di Liskov è violato).



Nell'esempio S2 invece si tiene conto di ciò che il rettangolo e il quadrato hanno in comune e l'ereditarietà dinamica è rispettata.

Programmazione Generica

È possibile rendere una classe parametrica tramite l'inserimento di uno o più metodi/attributi parametrici. Il comportamento della classe non dipende dal tipo specifico del parametro. Ogni classe generata da una classe generica non condivide con questa nessun legame di ereditarietà, ma costituisce una classe indipendente

Relazioni

La maggior parte delle classi e degli oggetti interagisce con altre classi ed altri oggetti. Tale interazione è possibile solo se tra le diverse entità sussiste una relazione. Di conseguenza, oltre a modellare le entità coinvolte, è necessario modellare anche le relazione tra queste entità.

Nella modellazione object-oriented le relazioni principali sono:

- **generalizzazione/ereditarietà** (Is a)
- **realizzazione** (implements): usata dalle interfacce
- **associazione**
    - generica
    - aggregazione (has): relazione non forte in quanto le classi parte hanno significato anche senza che sia presente la classe contenitore, inoltre le classi parte hanno un tempo di vita slegato d quello della classe contenitore (es. un museo contiene quadri e statue)
    - composizione (has subpart): relazione forte in quanto le parti hanno senso solo se in presenza della classe contenitore, inoltre le classi parte hanno tempo di vita dipendente da quello della classe contenitore e vengono distrutte alla distruzione del contenitore (es. un animale ha quattro zampe)
- **dipendenza**
    - collaborazione (uses)
    - relazione **istanza - classe**
    - relazione **classe-metaclasse**

Queste relazioni possono essere interpretate secondo lo schema cliente/fornitore.

Processo di sviluppo orientato agli oggetti

Si parte da un assunto di base: il mondo è composto di oggetti. Un sistema software rappresenta una porzione del mondo reale (o virtuale) ed è composto anch'esso di oggetti che interagiscono tramite scambio di messaggi.

- **Analisi orientata agli oggetti**: ha come scopo la modellazione di una porzione del mondo reale. Ogni classe descrive una categoria di oggetti.

- **Progettazione orientata agli oggetti**: ha come scopo la modellazione della soluzione. In questa fase vengono introdotti gli *oggetti di programmazione* (algoritmi, strutture dati...). Ogni classe descrive un tipo di dato differente.
- **Programmazione orientata agli oggetti**: ha lo scopo di realizzare la soluzione tramite l'utilizzo di linguaggi di programmazione OO e di sistemi run-time. Ogni classe descrive l'implementazione di tipo di dato.

Struttura a livelli di un'applicazione

Ciò che si trova tra quello che vede l'utente ed i dati fisicamente scritti in memoria può essere suddiviso in vari livelli:

- **Presentation Manager**: si occupa dell'interazione con l'utente tramite un'interfaccia (GUI o a caratteri)
- **Presentation Middleware**: software che permette di trasferire i dati da una interfaccia all'applicazione e viceversa (es. software di emulazione di terminali numerici, web browser + web server + HTTP, ...)
- **Presentation Logic**: gestisce l'interazione con l'utente a livello logico (accettazione dei dati, parziale validazione dei dati, gestione errori...)
- **Application Logic**: logica dell'applicazione (talvolta separata in più livelli comunicanti per mezzo di un Application Middleware che gestisce la comunicazione tra sue componenti della stessa applicazione)
- **Data Logic**: gestisce la persistenza a livello logico (consistenza dei dati, gestione degli errori, gestione file/database)
- **Database Middleware**: software che permette di trasferire i dati dall'applicazione ad un gestore di persistenza e viceversa (es. le richieste SQL dall'applicazione al DBMS, i dati dal DBMS all'applicazione).
- **Data Manager**: gestisce fisicamente i dati dell'applicazione tramite l'uso di file o di database

I livelli centrali costituiscono l'applicazione vera e propria. Un'applicazione dovrebbe essere il più generale possibile e comunicare in maniera logica con il mondo esterno in modo da essere resistente ai cambiamenti.

.NET

Tecnologia COM

Nasce come standard indipendente dalla piattaforma. Specifica quali siano i requisiti che i linguaggi COM devono implementare al fine di permettere l'interazione tra oggetti come:

Reference counting

La specifica COM impone che ogni oggetto rimanga "in vita" fintanto che c'è qualcuno che ne ha accesso ossia ogni oggetto è responsabile della gestione della propria memoria e si occupa di liberarla quando il reference count è zero (Questo porta a problemi quando due o più oggetti sono referenziati circolarmente)

Composizione e delega

Tale tecnologia prevede l'ereditarietà sia gestita solo attraverso composizione e delega (HAS A piuttosto che IS A) che in accoppiata con le interfacce permette una maggiore flessibilità ed estendibilità del codice.

DLL hell

La locazione di ciascun componente è salvata nel Windows registry, ciò significa che può esistere una sola versione di ogni componente. Questa limitazione crea problemi per il deployment.

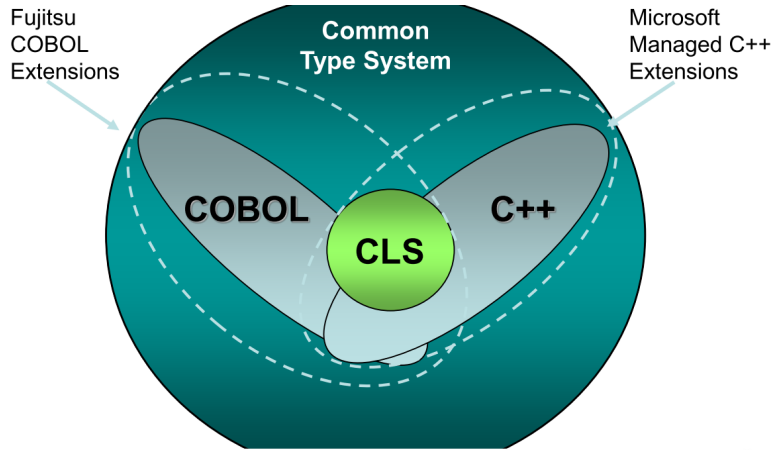
.NET

È un ambiente di esecuzione associato ad una libreria di classi indipendente da COM (vediamo .NET non COM), ma fortemente integrato con COM (non possiamo portare .NET fuori da COM). È un ambiente completamente OO, qualsiasi entità è un oggetto, l'ereditarietà è pienamente integrata (ereditarietà e composizione) ed è presente un garbage-collector (ma a differenza di COM non si basa su Reference Counting). La gestione degli errori viene attuata attraverso le eccezioni che possono essere generate dal codice in esecuzione o dall'ambiente di runtime: concetti universali sono lancio (throw), cattura (catch) ed esecuzione codice di uscita da un blocco controllato (finally) disponibili in tutti i linguaggi con sintassi diverse.

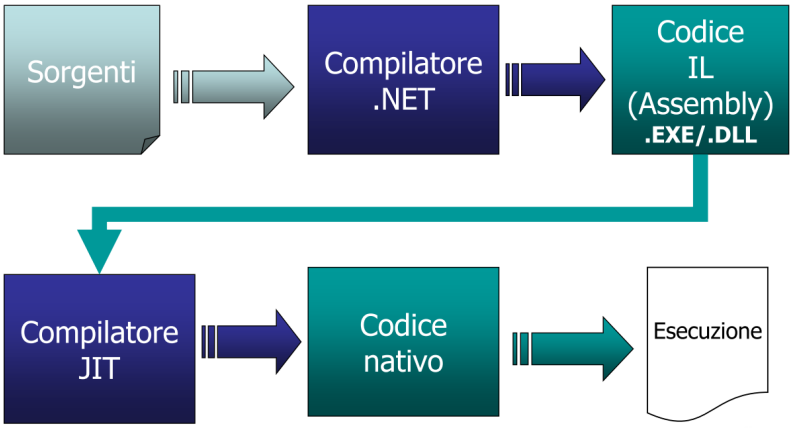
Tutti i linguaggi che stanno sopra il .NET sono fortemente tipizzati (tutti i tipi non sono gestiti indipendentemente dal linguaggio ma sono specificati dalla piattaforma .NET). Tutte le caratteristiche di .NET sono presenti in tutti i linguaggi, declinate secondo le caratteristiche di quel particolare linguaggio. Questo rende possibile scrivere varie parti dello stesso programma in linguaggi diversi e estendere classi scritte in un altro linguaggio (.NET si occuperà poi di tradurre queste caratteristiche in quelle di COM).

.NET è un'implementazione di **CLI** (Common Language Infrastructure), un ambiente di esecuzione dello standard ECMA-335 che permette ad applicazioni scritte in più linguaggi di alto livello di essere eseguita su sistemi differenti senza bisogno di riscrivere l'applicazione.

- È presente un linguaggio intermedio (Microsoft Intermediate Language o **(MS)IL**) per rendere portabile il codice (facile da tradurre in linguaggio macchina) e renderlo complicato da modificare (difficile da ingegnerizzare al contrario).
- L'ambiente di esecuzione è detto **CLR** (Common Language Runtime)
- È presente un sistema di tipi comune **CTS** (Common Type System)
- Sono presenti delle regole **CLS** (Common Language Specification) per permettere ad un linguaggio di interoperare all'interno di .NET (CLS è sottoinsieme di CTS)



Il **compilatore .NET** si occupa di tradurre in sorgente il codice IL, che può presentarsi sotto forma di libreria o di eseguibile. Una volta lanciato il programma questo linguaggio intermedio (molto simile al linguaggio macchina) viene compilato da un **compilatore JIT** in codice nativo che viene poi eseguito dalla macchina fisica.



L'unità minima per la distribuzione ed il versioning .NET è detta "Assembly" ed è composta da codice e metadati (descrizione dell'assembly, descrizione dei tipi contenuti all'interno, risorse).

I metadati sono usati per la compilazione (anche condizionale) JIT, dagli ambienti di sviluppo, da tool di analisi, dagli sviluppatori per eseguire l'analisi del contenuto (reflecion - introspezione). Con queste caratteristiche è possibile avere l'esecuzione contemporanea di versioni diverse dello stesso assembly e di avere condivisione di componenti.

Tipi di dato

All'interno di .NET si hanno sia tipi valore (mantenuti sullo stack) che tipi riferimento (mantenuti sullo heap). I tipi valore comprendono i tipi primitivi, le struct e gli enum, mentre i tipi riferimento sono tutte le altre classi. Sia i tipi primitivi che i tipi riferimento derivano da Object ("ValueType" o "ReferenceType"), pertanto possono essere usati interscambiabilmente. È possibile convertire un tipo in un altro:

- **Conversioni implicite** (non generano eccezioni)
    - conversione numerica: il tipo di destinazione deve essere in grado di contenere senza perdite tutti i valori ammessi dal tipo di partenza
    - up cast: si basa sul principio di sostituibilità
    - boxing: l'oggetto boxed è un clone indipendente
    - definita dall'utente tramite la creazione di un nuovo operatore con il modificatore *implicit*
- **Conversioni esplicite** (possono generare eccezioni)
    - conversione numerica: non è detto che il tipo di destinazione sia in grado di contenere senza perdite tutti i valori ammessi dal tipo di partenza
    - down cast
    - unboxing
    - definita dall'utente tramite la creazione di un nuovo operatore con il modificatore *explicit*

In generale è consentita solamente l'ereditarietà semplice, ma è possibile implementare più interfacce (possono farlo sia i tipi valore che i tipi riferimento, sebbene una interfaccia sia sempre considerata un tipo riferimento). È possibile implementare un'interfaccia sia in maniera implicita (alla Java) sia in maniera esplicita, in questo caso per poter invocare il metodo contenuto all'interno dell'interfaccia occorre effettuare il cast dell'oggetto dal suo tipo originale al tipo dell'interfaccia.

Metodi

Modificatori dei metodi:

- **virtual**: se un metodo è marcato come virtuale la sua implementazione può essere cambiata da una sottoclasse (override) e viene effettuato a run-time un controllo sul tipo dell'oggetto tramite cui è possibile risalire a quale sia l'implementazione specifica di quel metodo (controllo all'interno della VM).

  Di default in .NET i metodi **non** sono virtuali: per B-> A, se un oggetto B viene assegnato ad una variabile A, allora viene eseguito il codice di A
- **abstract**: un metodo astratto non contiene implementazione ed è implicitamente virtuale
- **override**: i metodi marcati come override forniscono una nuova implementazione al metodo ereditato. Perché un metodo sia sovrascrivibile è necessario che la classe base lo dichiari come abstract o virtual
- **sealed**: il marcatore sealed impedisce alle sottoclassi di effettuare l'override di quel particolare metodo

Tipi di parametro:

- **In** (*default* in C#):
    - L’argomento deve essere inizializzato prima di essere passato al metodo
    - L’argomento viene passato per valore (per copia).

      Nel caso si tratti di un tipo valore (value type) viene passata una copia dell'intero oggetto ed eventuali modifiche sull'oggetto non hanno effetto sul chiamante.

      Nel caso si tratti di un tipo riferimento (reference type) viene passata una copia del riferimento all'oggetto ed eventuali modifiche sull'oggetto referenziato hanno effetto anche per il chiamante mentre modifiche sul riferimento non hanno effetti sul riferimento originale
- **In/Out** (*ref* in C#):
    - L’argomento deve essere inizializzato prima di essere passato al metodo
    - L’argomento viene passato per riferimento.

      Nel caso si tratti di un tipo valore (value type) eventuali modifiche sono visibili al chiamante (in quanto agiscono sull'oggetto originale)

      Nel caso si tratti di un tipo riferimento (reference type) hanno effetto sia modifiche sull'oggetto sia modifiche sul riferimento (in quanto viene passato l'indirizzo del riferimento originale)
- **Out** (*out* in C#):
    - L’argomento può non essere inizializzato prima di essere passato al metodo
    - L’argomento deve essere inizializzato all'interno del metodo
    - L’argomento viene passato per riferimento (come nel caso di In/Out), perciò le modifiche sull'oggetto hanno effetto sul chiamante

Costruttori

I **costruttori di istanza** provvedono all'inizializzazione dei campi di un nuovo oggetto, i **costruttori di tipo** provvedono all'inizializzazione dei campi statici quando la classe viene caricata in memoria. In generale il costruttore di tipo è dichiarato static (può accedere esclusivamente ai membri statici), implicitamente private, senza argomenti e va definito solo se strettamente necessario, ovvero se i campi statici della classe non possono essere inizializzati in linea oppure devono essere inizializzati solo se la classe viene effettivamente utilizzata.

Se si verifica un'eccezione nel costruttore e questa non viene gestita nel costruttore stesso, nel caso di costruttori di istanza nessun problema, nel caso di costruttori di tipo la classe non è più utilizzabile.

Interfaccia vs Classe astratta

Interfaccia	Classe astratta
Può "ereditare"da 0+ interfacce	Può "ereditare" da 0+ interfacce, da 0+ classi (astratte e/o concrete)
Non può essere istanziata	Non può essere istanziata
Non può contenere uno stato	Può contenere uno stato (comune a tutte le sottoclassi)
Non può contenere attributi membro e metodi statici (a parte eventuali costanti comuni)	Può contenere attributi membro e metodi (e proprietà ed eventi) statici
Non contiene alcuna implementazione	Può essere implementata completamente, parzialmente o per niente
Le classi concrete che la implementano devono realizzare tutte le funzionalità	Le classi concrete che la estendono devono realizzare tutte le funzionalità non implementate e possono fornire una realizzazione alternativa a quelle implementate
Se si aggiungesse un metodo a un'interfaccia già in uso, tutte le classi che implementano quell'interfaccia dovrebbero essere modificate	Quando si aggiunge un metodo a una classe astratta già in uso, è possibile fornire un'implementazione di default, in modo tale da non dover modificare le sottoclassi
Non può gestire la creazione delle istanze delle classi che la implementano	Può gestire la creazione delle istanze delle sue sottoclassi
La creazione deve essere effettuata dai costruttori delle suddette classi o da una classe non correlata, la cui unica funzionalità è la creazione di istanze di altre classi (classe factory)	La creazione può essere effettuata come per l'interfaccia, ma anche da un metodo statico della classe astratta (metodo factory)

Delegati ed eventi

I delegati sono oggetti che possono contenere il riferimento ad un metodo (sia esso associato ad un oggetto in particolare o sia esso associato ad un tipo), permettendo la creazione di funzionalità di callback (elaborazione asincrona, gestione degli eventi, elaborazione cooperativa). Ad ogni delegato è possibile associare una lista di metodi (*invocation list*, creata con l'operatore "+="') che all'atto dell'invocazione del delegato verranno invocati in sequenza e in maniera sincrona, nel caso tali metodi accettino un parametro esso sarà lo stesso per tutte le invocazioni (con le implicazioni descritte precedentemente), mentre se tali metodi ritornano qualcosa il valore ritornato dall'invocazione del delegato sarà il valore ritornato dall'ultimo metodo della lista.

Fornire un campo pubblico per la registrazione ad un delegato offre un accesso troppo ampio, di conseguenza si potrebbero rendere i delegati privati e creare dei metodi accessori/modificatori che disciplinino l'accesso agli stessi. Gli eventi forniscono nativamente esattamente questo comportamento (personalizzabile dal programmatore allo stesso modo delle altre proprietà).
Un evento può essere scatenato sia dall'interazione con l'utente sia dalla logica del programma stesso e in generale si ritrovano:

- **Event sender**: oggetto/classe che scatena l'evento, in generale non conosce né chi riceverà l'evento né come questo reagirà alla ricezione

- **Event receiver**: oggetto/classe che desidera essere notificato dell'avvenimento dell'evento
- **Event handle**: metodo del receiver che viene invocato alla ricezione dell'evento

Ciclo di vita di un oggetto

Per prima cosa viene allocata la memoria necessaria ad ospitare l'oggetto, successivamente all'interno di quest'area di memoria l'oggetto viene inizializzato. A questo punto l'oggetto è funzionante. In caso l'oggetto non contenga risorse associate al sistema operativo o risorse che vadano rilasciate (file aperti, socket, connessioni varie...) la memoria occupata dall'oggetto può direttamente essere liberata, rendendola disponibile per altri usi. In caso invece l'oggetto possedesse risorse particolari essere dovrebbero essere chiuse (l'oggetto dovrebbe essere finalizzato) prima della deallocazione della memoria.

Garbage Collector

Ogni oggetto usato all'interno di un programma OO ha la necessità di utilizzare un'area di memoria dove poter depositare il suo stato, tuttavia quando un oggetto non è più utilizzato vi è la necessità di liberare questa memoria. Il garbage collector fornisce un metodo per risolvere questo problema, migliorando la stabilità dei programmi (controlli sui memory leak...) anche se richiede ulteriori risorse di calcolo.

Sono possibili più strategie:

- **Tracing**: si determina quali oggetti siano raggiungibili, tutti gli altri possono essere eliminati. Definito un insieme di oggetti detti "radice", si passa ad analizzare iterativamente gli oggetti per determinare se siano raggiungibili. Un oggetto è raggiungibile se:
    - è una radice
    - è raggiungibile ricorsivamente partendo da una radice, ovvero esiste una catena di riferimenti che partendo da una radice conduce all'oggetto in questione
- **Refernce counting**: ad ogni puntatore viene associat un contatore che indica il numero di riferimenti a esso, la memoria può essere liberata quando il contatore raggiunge lo 0. In caso di riferimenti incrociati diventa inutile, e comunque aumenta l'occupazione di memoria
- **Escape analysis**: vengono spostati gli oggetti dallo heap allo stack, l'analisi è fatta a compile time per determinare quali oggetti abbiano la necessità di essere accessibili fuori dalla routine che li ha creati. Gli oggetti che si trovano sullo stack vengono deallocati automaticamente all'uscita della routine

Inizialmente il GC ipotizza che tutti gli oggetti siano garbage, scorrendo la memoria marca via via gli ogetti raggiunti per via diretta o indiretta. Terminata la scansione gli oggetti non marcati vengono eliminati. Fatto ciò la memoria viene ricompattata, modificando tutti i riferimenti agli oggetti sopravvissuti.

Finalizzazione e rilascio deterministico

La finalizzazione non è responsabilità del GC ma del programmatore. Tutte le risorse unmanaged vanno chiuse prima della distruzione dell'oggetto, meglio se deterministicamente. A questo proposito esiste il pattern dispose: ogni oggetto che gestisce una risorsa unmanaged implementa l'interfaccia IDisposable, e deve implementare il metodo Dispose.

Metadati ed introspezione

I matedati possono essere definiti come dei "dati che forniscono informazioni su altri dati". Per quanto riguarda la programmazione la **reflection** fornisce un metodo per esplorarli e trarne vantaggio, essa permette infatti di esaminare il contenuto di un assembly (nel caso di .Net, package/class nel caso di Java), istanziarne oggetti e chiamarne metodi. Queste caratteristiche permettono la creazione di librerie generiche per visualizzare dati, serializzazione/deserializzazione di dati per una comunicazione, o la manipolazione classi generiche (vedi Hibernate) introducendo, ad esempio, nuovi attributi a run-time oppure decidendo quale specifica classe istanziare a run-time.

La metaprogrammazione è una tecnica di programmazione in cui un programma è in grado di considerarne un altro come dati (reificazione) su cui è possibile lavorare, questo è possibile solamente se i metadati presenti a compile-time sono mantenuti a run-time.
Usando tecniche di metaprogrammazione è possibile, ad esempio, modificare/creare un assembly "on the fly".

Design Pattern

Ogni pattern descrive un problema ricorrente e descrive successivamente una soluzione allo stesso problema in modo che essa possa essere riutilizzata più volte.

I pattern possono essere divisi in tre categorie:

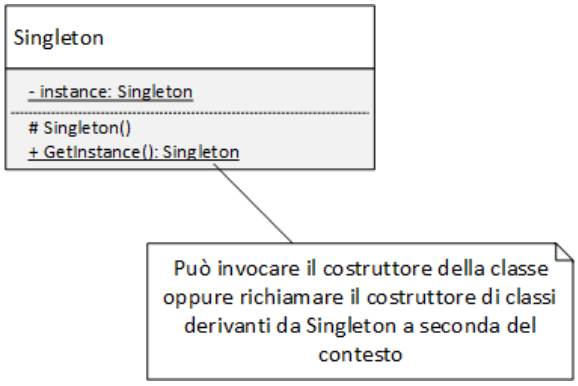
- **Pattern di creazione**: Risolvono problemi inerenti il processo di creazione di oggetti
- **Pattern strutturali**: Risolvono problemi inerenti la composizione di classi o di oggetti
- **Pattern comportamentali**: Risolvono problemi inerenti le modalità di interazione e di distribuzione delle responsabilità tra classi o tra oggetti

Pattern di creazione	*Pattern strutturali*	*Pattern comportamentali*
Abstract Factory	**Adapter**	Chain of Responsability
Builder	Bridge	Command
Factory Method	**Composite**	Interpreter
Prototype	**Decorator**	**Iterator**
Singleton	Facade	Mediator
	Flyweight	Memento
	Proxy	**Observer**
		State
		Strategy
		Template Method
		Visitor

Pattern Singleton

Assicura che una classe abbia una sola istanza e fornisce un punto di accesso globale a tale istanza. Tale classe deve tener traccia della sua singola istanza, garantire che non ne vengano create altre e fornire un modo per accedere all'istanza unica.

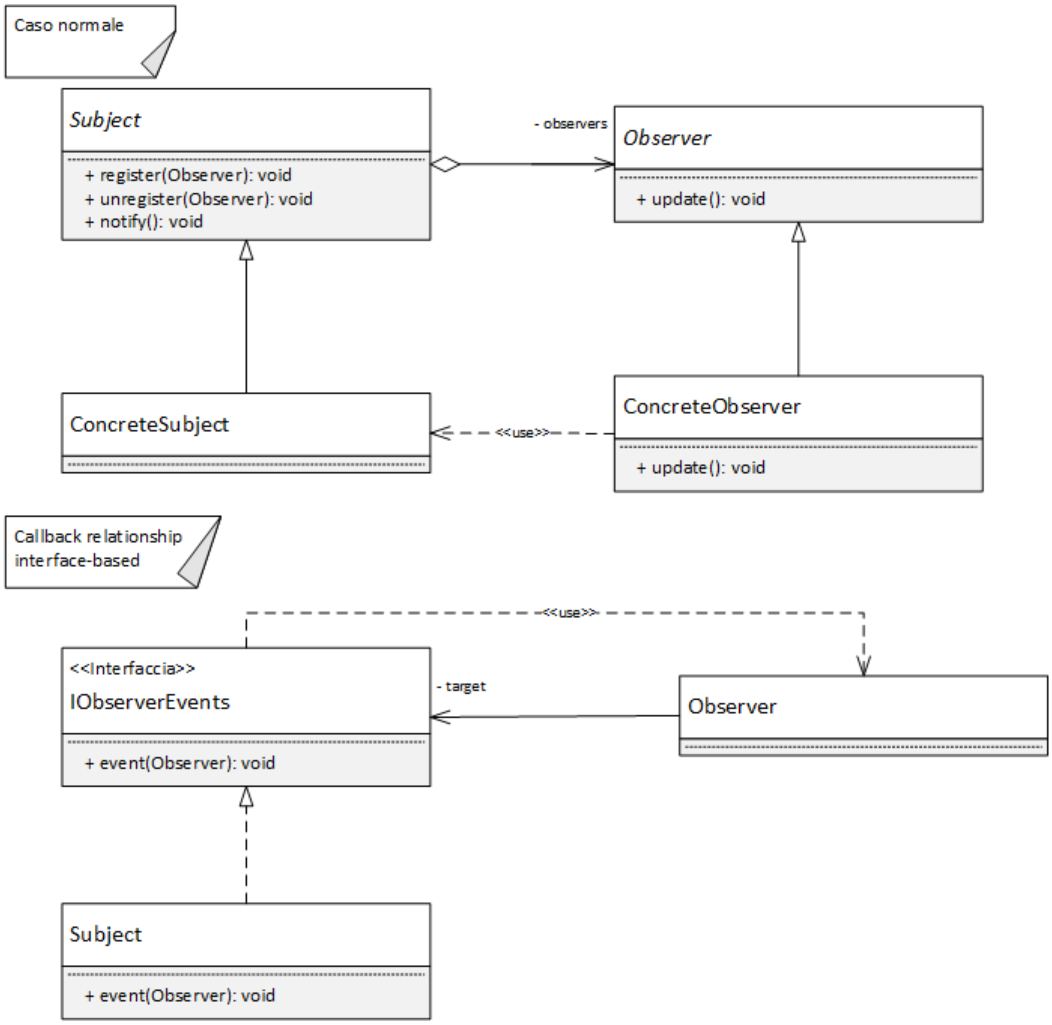
La classe in esame deve avere uno o più costruttori protected e può essere istanziata solamente attraverso il metodo GetInstance() che invocherà il costruttore corretto a seconda del contesto. Può implementare un numero arbitrario di interfacce.



Pattern Observer

Utile quando un oggetto (il **subject**) ha la necessità di notificare altri oggetti (**observer**) circa l'avvenimento di un qualche evento. Tre tipi di realizzazione:

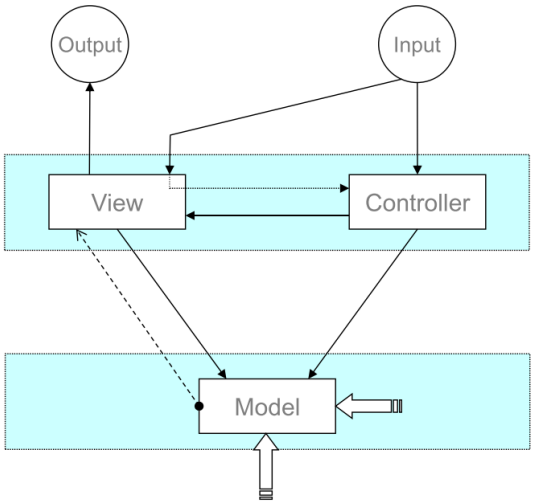
1. La soluzione sta nel creare una classe astratta che racchiude tutti gli observer, ed eventualmente una classe astratta che racchiuda tutti i subject (possono essere usate anche delle interfacce). Ogni oggetto che implementi questa classe dovrà registrarsi (passare un riferimento all'oggetto stesso) all'interno del subject nella lista di eventi corretta. Tramite questi riferimenti il subject potrà notificare l'avvenimento di un evento agli observer (attraverso un metodo notify).
2. Alternativamente alla prima si possono usare, nei linguaggi che li supportano, delegati o eventi per ottenere lo stesso risultato, ma senza i problemi legati all'accoppiamento delle classi.
3. Ulteriore alternativa al pattern observer è la callback relationship che si divide in:
    - class-based: viene inserito anche un riferimento al subject nell'observer in modo da avere una dipendenza reciproca tra i due componenti
    - interface-based: l'observer ha un riferimento all'interfaccia IObserverEvents che viene implementata dal subject. In questo modo il subject dipende dall'observer ma non viceversa.



Pattern Model/View/Controller (MVC)

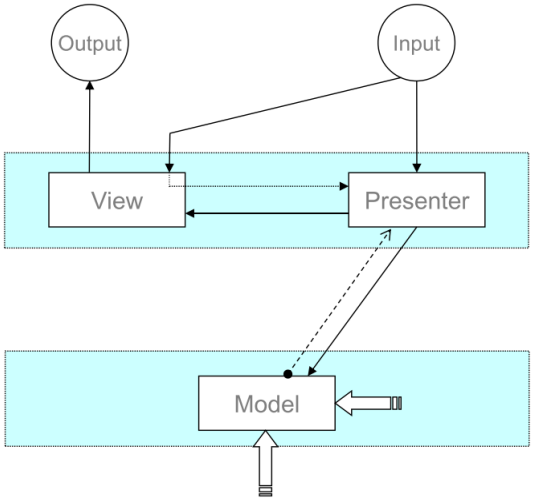
Utilizzato per suddividere le applicazioni grafiche in tre parti:

- **Model**: si occupa di gestire l'insieme dei dati e di rispondere alle interrogazioni su di essi. Risponde alle istruzioni di modifica dello stato e genera eventi al cambiamento dello stato. In generale può essere un subject del pattern Observer
- **View**: fornisce una vista dei dati mappandoli in oggetti visuali registrandosi presso il modello per essere notificato al cambiamento dello stato. In generale è un observer relativamente al pattern Observer.
- **Controller**: gestisce l'input dell'utente mappando le sue azioni in comandi



Pattern MVP

Presenta una soluzione diversa rispetto al pattern MVC che permette il riutilizzo delle view. In questo caso il model non interagisce direttamente con la view, ma lo fa per mezzo di un presenter che è associato a quella view.

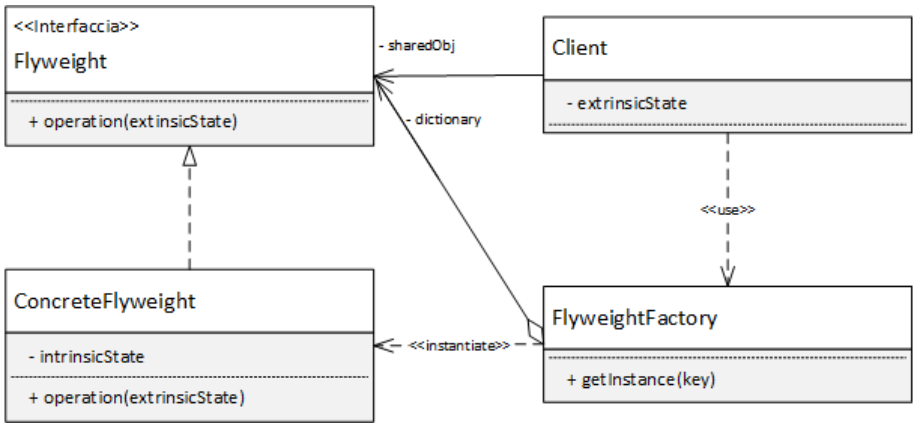


Pattern flyweight

Questo pattern descrive come si possa condividere oggetti a granularità molto fine (leggeri) in modo che il loro uso non sia troppo costoso. Un flyweight è un oggetto condiviso, ma dal punto di vista del cliente non deve notarsi la differenza da un oggetto non condiviso. Per assicurarsi che questi oggetti vengano creati e condivisi correttamente essi non devono mai essere creati autonomamente, ma sempre per mezzo di una factory.

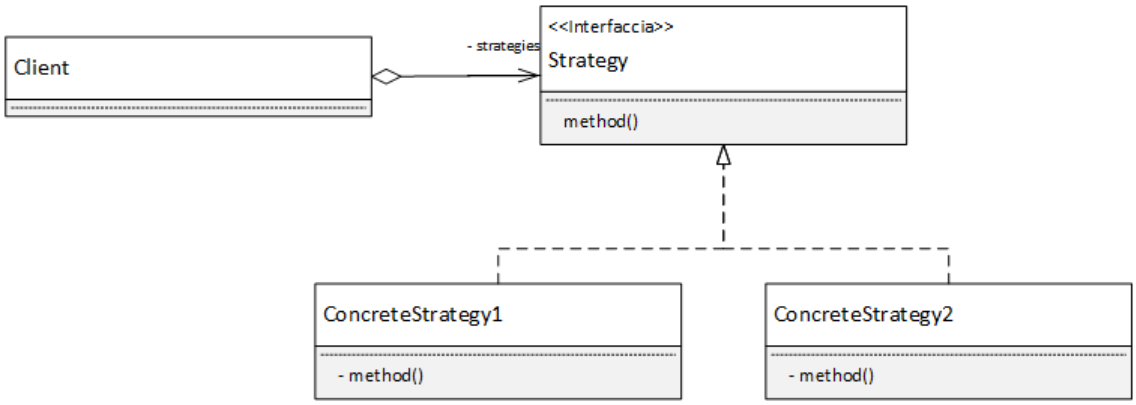
Viene effettuata la distinzione tra **stato intrinseco** non dipendente dal contesto di utilizzo e quindi condivisibile da tutti i clienti e **stato estrinseco** dipendente dal contesto di esecuzione e quindi non condivisibile. Lo stato intrinseco è memorizzato all'interno del flyweight, mentre lo stato estrinseco è a carico del cliente che lo usa.

Un esempio di utilizzo di questo pattern è la condivisione delle icone tra i vari clienti all'interno del sistema operativo.



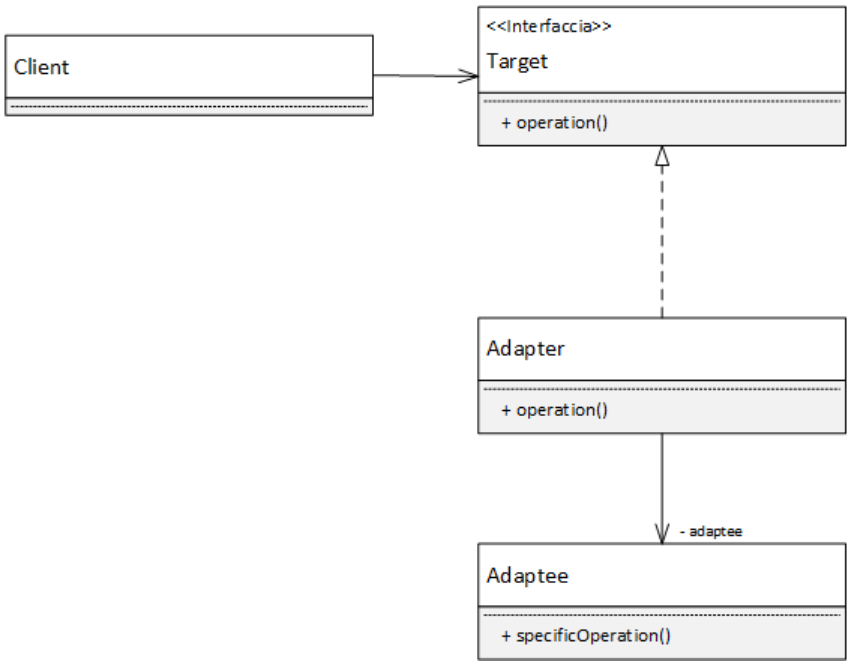
Pattern strategy

Utilizzato per definire un insieme di algoritmi tra loro correlati e renderli interscambiabili. Essi vengono incapsulati in una gerarchia di classi, che deve rispettare il principio di sostituibilità di Liskov, (oppure in classi che implementano la stessa interfaccia) e pertanto dal punto di vista del cliente sono identici nel loro scopo.



Pattern adapter

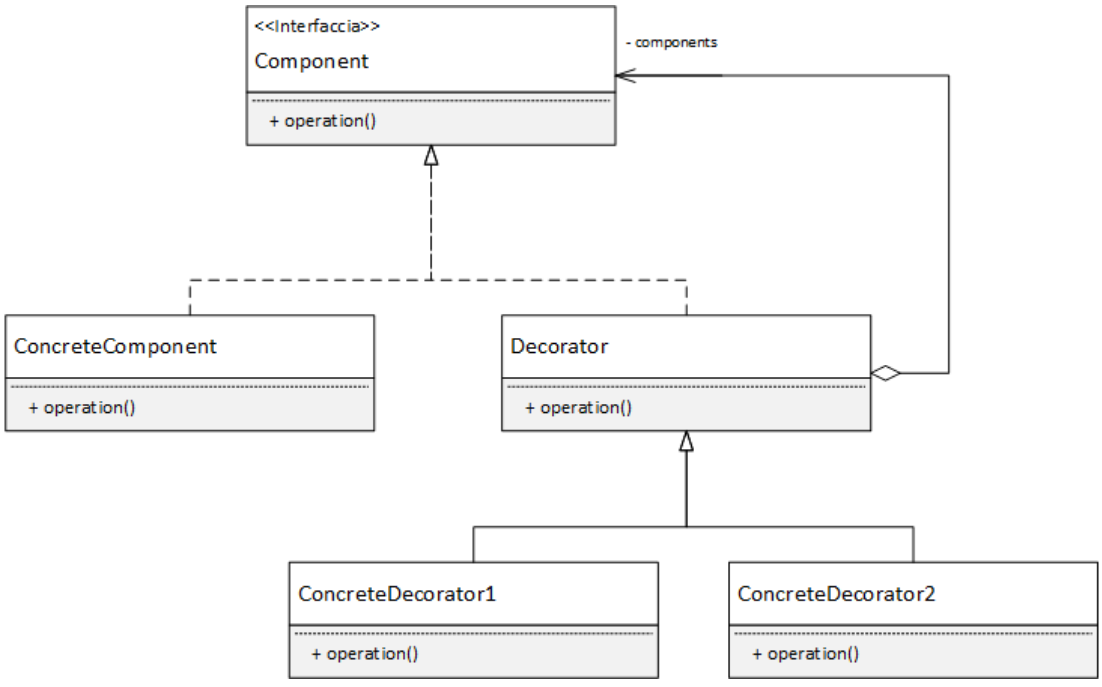
Utilizzato per convertire l'interfaccia di una classe in quella che si aspetta un cliente. Permette a classi che hanno interfacce incompatibili di lavorare insieme. Noto anche come **wrapper**, viene usato per riutilizzare classi esistenti adattandole a nuove interfacce.



Pattern decorator

Usato per aggiungere dinamicamente responsabilità ad un oggetto. È un'alternativa alla specializzazione da preferire quando le estensioni possibili sono talmente tante da rendere impossibile supportare ogni combinazione. Viene realizzato tramite una gerarchia di classi aventi come radice la classe astratta **Component**, la quale dichiara l'interfaccia comune a tutti gli oggetti a cui è possibile aggiungere responsabilità. Essa viene specializzata in:

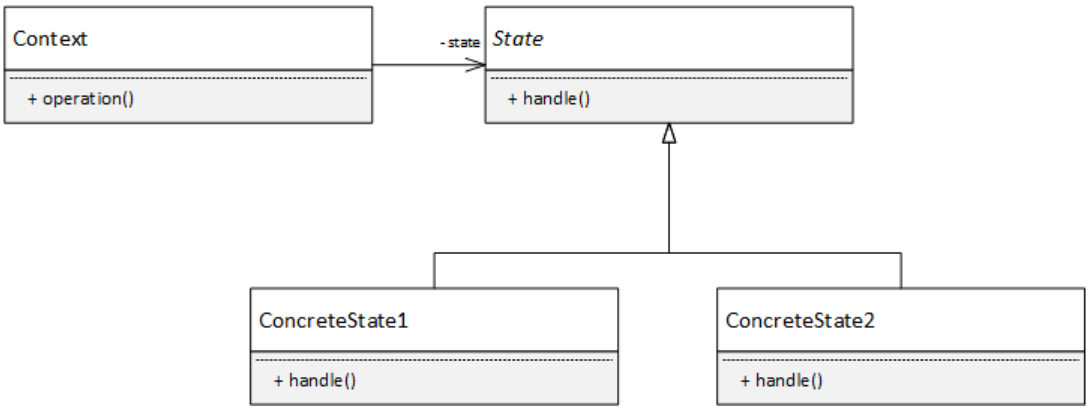
- **ConcreteComponent** definisce un tipo di oggetto al quale deve essere possibile aggiungere dinamicamente responsabilità
- **Decorator** (classe astratta) mantiene un riferimento ad un oggetto di tipo Component, definisce un'interfaccia comune per i vari decorator. Essa poi verrà specializzata in vari **ConcreteDecorator** che aggiungeranno responsabilità al component referenziato.



Pattern state

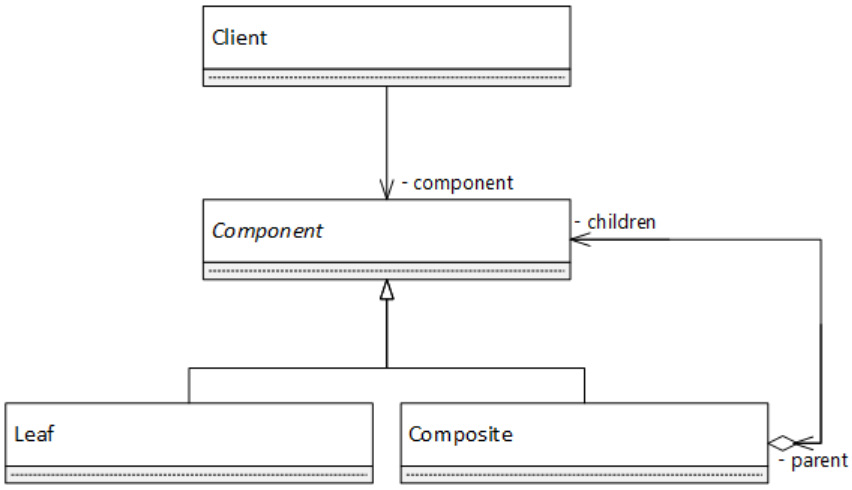
Pattern comportamentale che permette ad una classe di cambiare il proprio comportamento in base al suo stato. Permette di localizzare il comportamento relativo ad uno stato suddividendolo in base ad esso all'interno di classi separate che gestiscono la logica relativa alle transizioni da uno stato all'altro.

Permette anche di simulare l'ereditarietà multipla.



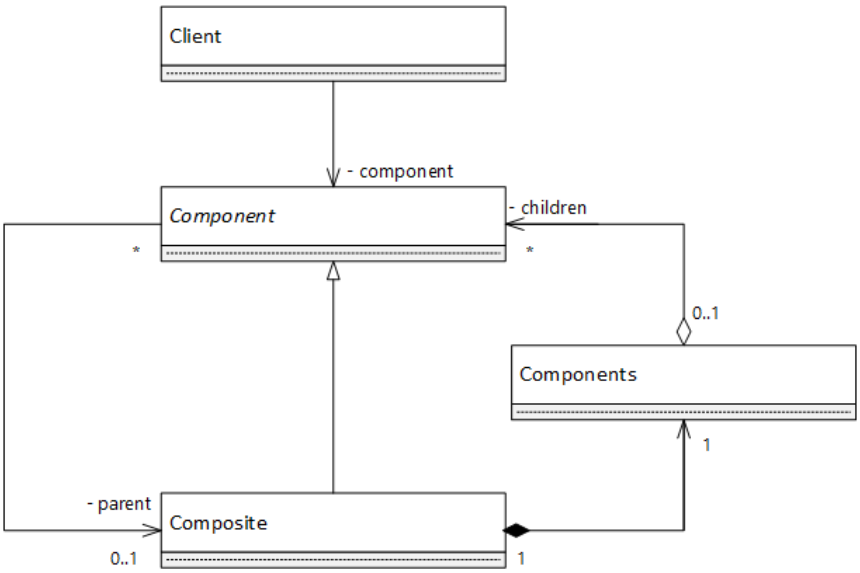
Pattern composite

Permette di comporre oggetti in una struttura ad albero per rappresentare una gerarchia contenuto-contenitore, dando la possibilità ai clienti di trattare in modo uniforme gli oggetti singoli e gli oggetti composti. Viene realizzata una classe astratta **Component** che dichiara un'interfaccia comune e realizza il comportamento di default, essa poi potrà venir specializzata in classi che rappresentano le foglie dell'albero (**Leaf**) ed in classi che invece ne rappresentano i nodi (**Composite**, contiene all'interno una collezione di **Component**).



I riferimenti che permettono di navigare l'albero possono essere contenuti sia all'interno dei nodi padre che all'interno dei nodi figlio. Nel primo caso è agevolato l'attraversamento top-down dell'albero, nel secondo è agevolato l'attraversamento bottom-up.

Per evitare la dipendenza ciclica:



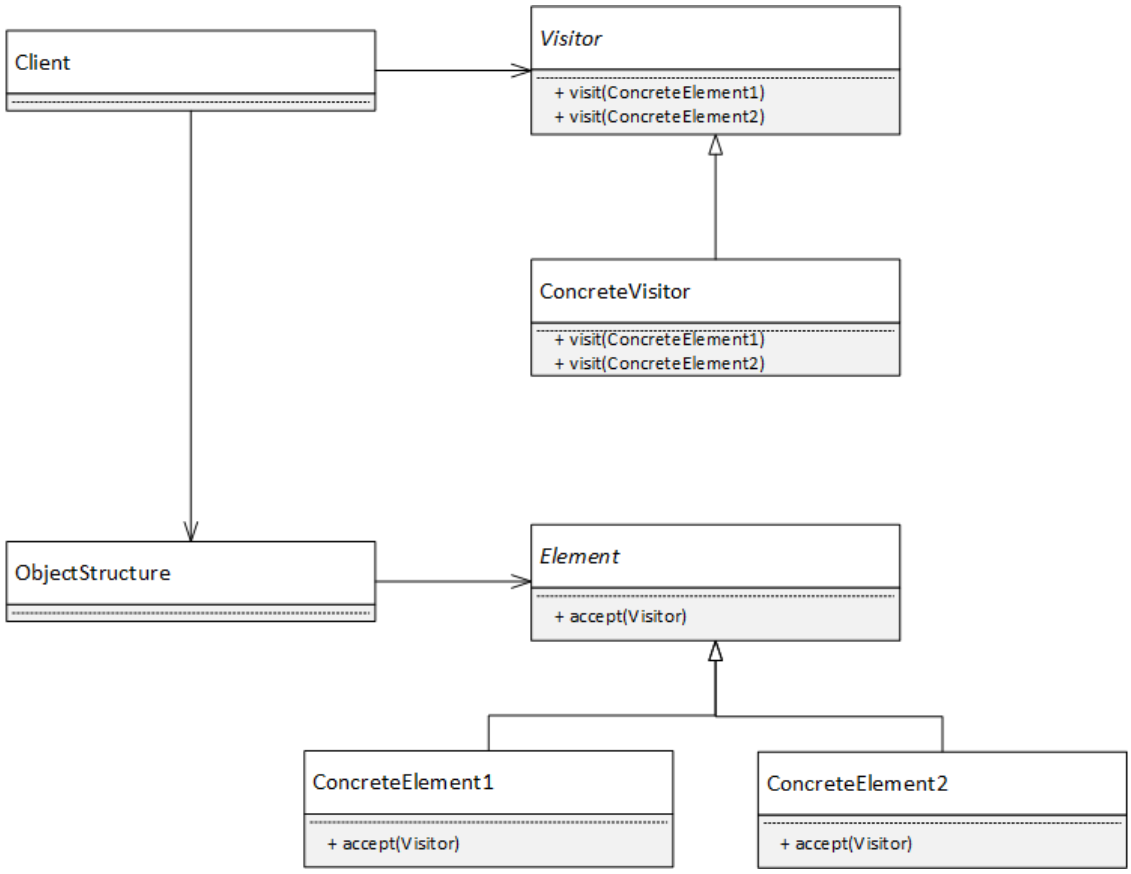
Occorre decidere cosa includere nell'interfaccia di **Component**:

- Component ha una fat interface che contiene tutte le operazione che possono essere effettuate sui figli (di qualunque tipo essi siano). Tuttavia è possibile che alcune di queste operazioni non abbiano significato in tutti i casi (ad esempio aggiungere un figlio ad una foglia). Il cliente di conseguenza è obbligato a conoscere le due interfacce diverse o a prepararsi a catturare delle eccezioni
- Component ha un'interfaccia ridotta che contiene solamente le operazioni fondamentali comuni a tutti i tipi di nodo. Questo implica una maggior sicurezza nell'uso dell'interfaccia, ma costringe il cliente a conoscere la differenza tra le interfacce di **Composite** e di **Leaf** per poter ad esempio inserire figli

Pattern visitor

Permette di definire una nuova operazione da effettuare sugli elementi di una struttura dati senza dover modificare le classi degli elementi coinvolti.

Le operazioni da eseguire vengono rimosse dalle classi che compongono l'abstract syntax tree e vengono inserite nella gerarchia dei Visitor. L'interfaccia/classe astratta **Visitor** deve definire un metodo specifico per ogni elemento da visitare (metodo visit che accetta come parametro un riferimento al nodo concreto associato a quell'operazione). Le classi derivate da Visitor forniscono un'implementazione del metodo relativo all'elemento su cui agiscono ed eventualmente possono possedere uno stato interno. Al contempo ogni elemento della struttura dati deve definire un metodo (accept) per accettare la visita del Visitor. Durante la visita della struttura dati per ogni nodo verrà invocato il metodo visit corretto, tuttavia per adempire al suo scopo ogni visitor ha la necessità di accedere allo stato degli elementi su cui opera.



Questo pattern facilita l'aggiunta di nuove operazioni poichè è possibile aggiungere nuove operazioni su una struttura esistente semplicemente aggiungendo un nuovo visitor concreto, rendendo non necessaria la modifica delle classi facenti parte della struttura. Al contempo, tuttavia, è difficile aggiungere una nuova classe alla struttura poichè questo comporta la modifica di tutti i visitor esistenti, questo comporta che la struttura dati debba essere stabile nel tempo.

Con il termine **double dispatch** si intende che l'operazione che deve essere effettuata dipende dal tipo di due oggetti (il visitor e l'elemento). Accept() è un'operazione di tipo double dispatch .

Qualità del design / Bad design

La qualità della progettazione dipende dalle priorità dell'organizzazione che ha commissionato un prodotto. Una buona progettazione potrebbe essere la più efficiente, così come la più economica o quella maggiormente mantenibile.
Se da un lato è difficile definire cosa sia un buon design, è relativamente semplice definire cosa sia un cattivo design:

- **Misdirection**: Il design potrebbe non riuscire a soddisfare i requisiti
- **Rigidità**: riguarda la difficoltà di modificare un software. Accade quando una singola modifica causa a cascata sempre più modifiche da apportare al codice già esistente
- **Fragilità**: riguarda la tendenza del software a rompersi in più punti contemporaneamente successivamente a delle modifiche, spesso in aree non logicamente collegate tra loro, inoltre ogni correzione causa ulteriori problemi
- **Immobilità**: riguarda l'inabilità di riusare del software già esistente a causa dell'enorme mole di dipendenze che si porta dietro
- **Viscosità**: accade quando ripetutamente anziché modificare correttamente il software si fa ricorso a delle scorciatoie, e porta ad una difficoltà sempre maggiore nel fare la "cosa giusta". La viscosità fa degradare il software molto più rapidamente di quanto farebbe normalmente

Questi problemi possono essere causati da:

- inabilità dei programmatori/progettisti
- mutamenti nelle tecnologie
- vincoli stringenti relativi al costo della manutenzione
- mutamenti nei requisiti nel tempo
- gestione impropria delle dipendenze tra i moduli software

Design Principle

Principi software

Principio zero o rasoio di Occam

Non bisogna introdurre concetti che non siano strettamente necessari / quel che non c'è non si rompe

Tra le varie soluzioni possibili occorre scegliere quella più semplice e quella che si porta dietro meno ipotesi/concetti di base e genera meno dipendenze. La semplicità iniziale va poi mantenuta lungo tutta la durata dello sviluppo.

Single Reponsability Principle

Una classe (e in generale un modulo, una funzione, ...) ha un unico scopo e deve portarlo a termine completamente e correttamente. Se una classe ha più responsabilità allora essa tende a diventare fragile poiché cambiamenti in una delle funzionalità vanno ad impattare potenzialmente anche sui clienti delle altre funzionalità. Questa situazione si risolve con il refactoring, dividendo le singole responsabilità in varie classi (una responsabilità per classe).

Dependency Inversion Principle

I vari moduli (o le varie classi) non devono dipendere l'un l'altro (concretamente) ma dipendere da delle astrazioni. Generalmente i moduli di basso livello contengono la maggior parte del codice e quindi sono quelli maggiormente soggetti ai cambiamenti, di conseguenza se i moduli di livello più alto sono accoppiati in maniera troppo stretta con essi le modifiche potrebbero propagarsi (portando a rigidità, fragilità, immobilità). Introducendo delle astrazioni intermedie (che in teoria non contengono codice e quindi sono molto stabili) invece si rompe l'accoppiamento proteggendo i moduli di alto livello dai cambiamenti e rendendoli riutilizzabili.

Questo principio permette anche di rompere le dipendenze transitive (A->B->C diventa A->IB<-B->IC<-C) e le dipendenze cicliche (A<->B).

Interface Segregation Principle

"Un cliente non deve essere forzato a dipendere da delle interfacce che non usa". Le "fat interfaces" creano una forma indiretta di accoppiamento tra i client e perciò vanno divise in tante interfacce quanti sono i servizi fondamentali.

Open/Closed Principle

Le entità software devono essere *aperte* a estensioni (aggiungendo nuove funzionalità), ma *chiuse* ai cambiamenti (le interfacce non devono essere modificate). Sebbene sembri una contraddizione, questo è reso possibile dall'uso delle interfacce: un modulo che usa un'interfaccia dipende da un'astrazione immutabile (in teoria), pertanto non dovrà mai essere modificato. Per aggiungere un nuovo comportamento è sufficiente creare una nuova classe che implementi quell'interfaccia e che contenga il codice relativo al comportamento desiderato (esempio del modulo copia che dipende dalle interfacce "IReader" e "IWriter" e non dalle implementazioni specifiche di qualche classe).

Principio di sostituibilità di Liskov

Afferma che una sottoclasse deve poter essere sostituita ad una superclasse senza che il cliente si accorga della differenza. Generalmente questo principio è violato quando un metodo viene ridefinito in una sottoclasse, ma si può rimediare a ciò sfruttando "Design by contract".

Design by contract

Ogni metodo ha un insieme di precondizioni, ovvero di requisiti minimi che devono essere soddisfatti dal cliente per poter invocare il metodo, ed un insieme di postcondizioni, ovvero di requisiti che devono essere soddisfatti dal metodo. Questi due insiemi costituiscono un contratto tra cliente e metodo. Quando un metodo viene ridefinito in una sottoclasse le precondizioni devono essere identiche o meno stringenti, mentre le postcondizioni devono essere identiche o più stringenti. Questo perché un cliente che invoca il metodo conosce il contratto definito a livello della classe base, quindi non è in
grado di soddisfare precondizioni più stringenti o di accettare postcondizioni meno stringenti. (Il quadrato è un rettangolo?)

Principi per l'architettura dei package

The Reuse/Release Equivalency Principle (REP)

Un componente riutilizzabile non può essere riutilizzato a meno che non venga gestito da un qualche sistema di rilascio. In generale un cliente non dovrebbe riutilizzare un elemento a meno che chi lo rilascia non garantisca di tener traccia dei numeri di versione e di mantenere anche delle versioni più vecchie per qualche tempo. Pertanto le classi vengono raggruppate in pacchetti per migliorarne le possibilità di riuso.

The Common Closure Principle (CCP)

All'interno di un pacchetto devono esser contenute solamente le classi che cambiano assieme. Questo tende a generare pacchetti di piccole dimensioni, agevolando i clienti finali nel riutilizzo degli stessi.

The Common Reuse Principle (CRP)

All'interno di un pacchetto dovrebbero essere contenute le classi che vanno usate assieme. Questo tende a generare pacchetti di grandi dimensioni, agevolando gli sviluppatori, ma complica la vita a chi dovrà poi usare i servizi offerti dal pacchetto.

Principi per le relazioni tra i pacchetti

The Acyclic Dependencies Principle (ADP)

Le dipendenze tra i pacchetti non devono essere cicliche. Se ci fosse una dipendenza ciclica l'albero delle dipendenze per la compilazione di un singolo pacchetto diverrebbe molto lunga, andando potenzialmente fuori controllo. Questo problema può essere risolto introducendo un nuovo pacchetto intermedio per spezzare il ciclo, oppure introducendo un'interfaccia.

The Stable Dependencies Principle (SDP)

Un pacchetto dovrebbe dipendere solo da pacchetti più stabili di lui stesso, ovvero a cui viene richiesto di cambiare molto di rado.

The Stable Abstractions Principle (SAP)

I pacchetti più stabili dovrebbero fornire delle interfacce su cui gli altri pacchetti possono fare affidamento, la stabilità in questo caso è relazionata alla quantità di lavoro necessario ad introdurre correttamente un cambiamento.

Discussione

Ipotizzando uno schema a livelli dei pacchetti, quelli che si trovano in cima sono necessariamente relativamente instabili e flessibili, mentre quelli che si trovano sul fondo sono stabili a causa della difficoltà di introdurre cambiamenti. Tuttavia, in base all'open/close principle, essi possono essere estesi con relativa facilità se essi sono pacchetti "astratti". Pertanto un'applicazione diventerebbe composta da pacchetti instabili, che però sono semplici da modificare, e da pacchetti stabili, che però sono semplici da estendere.

Progettazione di dettaglio

Durante la progettazione di dettaglio è necessario definire

- **Tipi di dato** che non sono stati definiti nel modello OOA
- **Navigabilità delle associazioni** tra classi e relativa implementazione. Riguarda la possibilità di spostarsi *efficientemente* da un oggetto di origine ad un oggetto di destinazione. Una relazione può essere sia monodirezionale che bidirezionale, in questo ultimo caso va effettuato un controllo sulla consistenza delle strutture dati. Le associazioni con molteplicità 1..1 o 0..1 possono essere risolte semplicemente aggiungendo un attributo membro, per le associazioni con molteplicità 0..* oppure 1..* occorre far ricorso ad una classe contenitore, che possono essere distinte:
    - dal modo con cui contengono gli oggetti:
        - per valore: nel contenitore viene memorizzata una copia del valore e la distruzione del contenitore implica la distruzione dei valori contenuti (gli oggetti originali possono sopravvivere)
        - per riferimento: il contenitore contiene un riferimento all'oggetto, pertanto la distruzione del contenitore non implica direttamente la distruzione degli oggetti contenuti
    - dall'omogeneità/eterogeneità degli oggetti contenuti
- **Strutture dati** necessarie per l'implementazione del sistema
- **Operazioni** necessarie per l'implementazione del sistema
- **Algoritmi** che implementano le operazioni
- **Visibilità di classi**
- **Risolvere i casi di ereditarietà multipla** tramite composizione e delega oppure tramite l'uso di interfacce
- **Miglioramento delle prestazioni**

Identificazione degli oggetti

Nel caso un oggetto si trovi interamente in memoria, nello spazio di indirizzamento dell'applicazione, esso può essere identificato univocamente tramite il suo indirizzo (logico) in memoria.

Nel caso invece si abbia a che fare con sistemi distribuiti o con dei database occorre associare ogni oggetto ad un identificatore univoco persistente tramite cui sia possibile risalire all'oggetto stesso. Questo identificatore può essere generato automaticamente (come nel caso dei campi autoincrementati dei DB) e fa parte degli attributi dell'oggetto da trovare.

Esempio tecnologia COM

La tecnologia COM permette ad un'applicazione di trovare, caricare e utilizzare run-time i componenti necessari per la sua esecuzione. Ogni componente è memorizzato in una DLL (Dynamic Link Library), che può trovarsi sia in locale che in remoto. Di conseguenza quando l'applicazione ha bisogno di un componente, il sistema deve essere in grado di localizzare la DLL che contiene quel particolare componente. Poichè non è possibile sfruttare il path o un indirizzo fisico ma occorre utilizzare un meccanismo di indirizzamento logico realizzato tramite degli identificatori globali (GUID) di vario tipo, la cui corrispondenza con gli identificatori fisici è mantenuta all'interno del registro di Windows. Disponendo del GUID corretto è possibile chiedere al sistema di istanziare un componente a run-time.

Version Control System (VCS)

Questi sistemi si occupano di tener traccia delle varie versioni di uno o più documenti, rendendo possibile la coesistenza di versioni multiple dello stesso documento o di effettuare un rollback ad una versione pecedente. Inoltre permette di procedere in parallelo su differenti aspetti della redazione di documenti (a prescindere dal tipo di documento) incrementando la produttività.

Sono presenti alcuni concetti chiave:

- **Progetto**: insieme di file (di qualsiasi tipo) gestito dal VCS
- **Repository**: rappresenta il luogo in cui i file, con la loro storia, vengono salvati. Tipicamente è costituito da un server remoto, ma nulla vieta la costruzione di un repository locale
- **Cartella di lavoro**: contiene una *copia* locale del contenuto di un repository. Concettualmente è una sandbox in cui lo sviluppatore può precedere con il proprio lavoro. Si definisce **check out** l'operazione che consiste nella copia dei dati dal repository alla cartella di lavoro, mentre si definisce **check in** l'operazione inversa.
- **Revisioni**: ogni volta che viene fatto il "check in" di un file ne viene creata una nuova versione, identificata da un valore incrementale. Poichè generalmente le modifiche sono relativamente piccole per efficienza vengono salvate le differenze tra la nuova versione e la vecchia. Se da un lato questo ottimizza l'occupazione dello spazio, dall'altro rende più complesse le operazioni di "check out" e di "check in"
- **Ramificazioni** (branch): consiste nella creazione di due copie coesistenti di uno o più file. Per evitare sovrascritture dei dati durante il "check in" sono possibili due strategie:
    - **Modello Lock-Modify-Unlock**: il repository consente ad un solo utente alla volta di modificare la stessa risorsa, che risulta pertanto bloccata per gli altri utenti. Quando l'utente ha finito con le modifiche viene effettuato il "check out" e la risorsa viene sbloccata. Questa tecnica potrebbe causare una linearizzazione del lavoro non necessaria in quanto rende impossibile a due utenti di lavorare contemporaneamente sulla stesso file ma in aree differenti. Inoltre può venir generato un falso senso di sicurezza: ad esempio l'utente A modifica il file f1, e l'utente B modifica il file f2. Singolarmente le due modifiche non presentano conflitti, ma, poiché f1 dipende da f2, il sistema smette di funzionare correttamente.
    - **Modello Copy-Modify-Merge**: non vengono introdotti dei lock, ogni volta che vengono effettuate delle modifiche ad una risorsa viene effettuato il merge tra le due versioni.
- **Merging**: consiste nell'unire due rami prima separati per ottenere un nuovo unico ramo. L'unione di due rami può avere successo o meno. Perché il merge abbia successo è necessario che non vi siano conflitti all'interno dei file. Un **conflitto** è rappresentato da una situazione in cui i due rami vanno a modificare la stessa porzione del documento originale, di conseguenza il VCS non è in grado di comprendere quale sia la versione definitiva e chi mantiene il repository deve intervenire a mano. I conflitti rilevati dai VCS sono puramente sintattici e non semantici, perciò è possibile che due modifiche sintatticamente corrette (per il VCS, non per il codice del programma ad esempio) portino ad un conflitto, ad esempio il cambiamento della firma di una funzione non crea conflitto sintattico ma rende impossibile la compilazione del codice che sfrutta la vecchia firma.

I VCS possono essere sia centralizzati sia distribuiti. Nel secondo caso il repository centrale deve sincronizzarsi con i vari client che posseggono una copia locale del repository. Un sistema distribuito inoltre rende più rapide operazioni comuni come commit e ricerca nella cronologia poiché non sussiste la necessita di comunicare con il server centrale. Quando si vuol far sì che le informazioni contenute nella propria copia locale del repository vengano condivise occorre effettuare un'operazione di "**push**" (l'operazione duale e detta "**pull**").

I VCS distribuiti hanno sia pro che contro. Da un lato permettono di lavorare efficientemente offline in una sandbox locale, dall'altro non esiste un concetto forte di "ultima versione" (si può considerare quella contenuta all'interno del repo centrale come l'ultima versione stabile) e si perde la nozione di numeri di revisione in favore dell'uso di "tag" descrittivi.